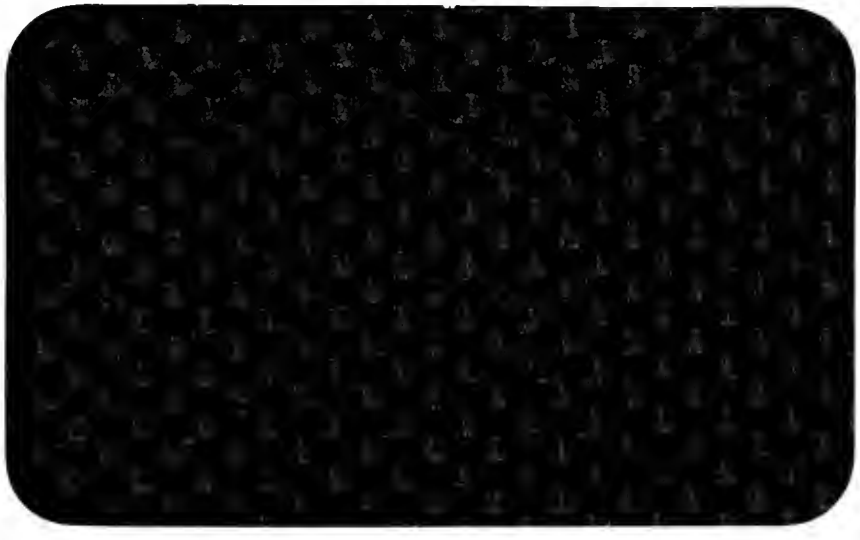
FAST PROBABILISTIC TECHNIQUES FOR PARALLEL ADDITION, PARALLEL COUNTING AND THE PROCESSOR IDENTIFICATION PROBLEM

By

Paul G. Spirakis
November 1984

Technical Report #144
Ultracomputer Note #80



FAST PROBABILISTIC TECHNIQUES FOR DYNAMIC PARALLEL ADDITION, PARALLEL COUNTING AND THE PROCESSOR IDENTIFICATION PROBLEM

By

Paul G. Spirakis
November 1984

Technical Report #144
Ultracomputer Note #80

Fast Probabilistic Techniques for Dynamic Parallel Addition, Parallel Counting, and the Processor Identification Problem

Paul G. Spirakis[1]

Courant Institute, N.Y.U.
251 Mercer Street
New York, N.Y. 10012

Ultracomputer Note #80

November, 1984

ABSTRACT

We consider here three problems in synchronous parallel computation, with the common characteristic that the shared memory available to the processors is restricted. The first is the problem of *dynamic parallel addition*: n processors are given each keeping a number x_i, $i = 1,\ldots,n$. We know that only $m < n$ of the x_i 's are nonzero but we don't know in advance which processors have the nonzero numbers. We show how to compute the sum of the x_i's in O(log m) *expected* parallel time and O(m) shared memory in the concurrent read – concurrent write model of parallel computation, through a probabilistic algorithm. We then consider the related problem of *processor identification*: n processors are given, each keeping either a 0 or a 1. We want each processor at the end to know which are the processors with the 1's. The shared memory available is very small (say 1 location), and can store only O(n) size numbers. To solve this problem O(n) time is required in a parallel read-exclusive write (or even concurrent write) PRAM, even if O(n) shared memory is available.

We show how to do it in O(n/logn) parallel time, if we use a weak version of a (more powerful) model of parallel computation, the so-called *paracomputer*. Our algorithm is again probabilistic. For this algorithm, we use combinatorial ideas developed by Erdos and Renyi [ER,63] . Finally, we show how to do *approximate parallel counting* of O(n) units in O(1) parallel time, by using O(n/logn) shared memory, and an n-processor concurrent read - concurrent write machine.

---

[1]This work was supported in part by the NSF Grant MCS 83-00630.

1. Introduction

We consider here three related problems in synchronous parallel computation. The models of parallel machines we use are (a) the W-RAM (see e.g. [SV,80] , [G,78]) and (b) the paracomputer (see [S,80]). Both models assume the presence of a (potentially) unlimited number of processors with (potentially unlimited) local memory in each processor. In both models, processors are capable of doing independent probabilistic choices on a fixed input. (This was first used by [R,82a,b] and [V,83]). W-RAM is like the P-RAM of [W,79] and [FW,78] in the sense that different processors can read the same memory location at the same time. However, W-RAM is stronger than P-RAM, because it allows simultaneous access to the same memory location for write operations (concurrent read - concurrent write, CRCW). In the case of a simultaneous write attempt, exactly one processor succeeds. We make no assumption of which one succeeds, but we assume that the failed ones are notified. This can be easily implemented having processors read the result of the "write". The paracomputer model also allows simultaneous writes to the same memory location, by permitting the simultaneous execution of many fetch-and-add (FA) instructions on the same memory location. (See, e.g. [GLR,80]). The effect of simultaneous actions by the processors is as if the actions occurred in some (unspecified) serial order. The semantics of an individual FA are as follows: Let v be a shared memory address and e a value. Then, if processor P executes x ← FA(v,e) alone (where x is a local variable of P), what happens is that (a) x is assigned the contents [v] of v, and (b) [v] ← [v] +e is executed. The logical order is first (a), then (b), but each FA takes constant time. The logical effect of many FA operations on the same shared variable is as if these operations occurred in some (unspecified) serial order. In the paracomputer, the simultaneous execution of many FA takes 1 step.

In this paper we use a *weak* version of a paracomputer: simultaneous FA are allowed *only if* all FA's try to update v by using *the same value* e. We call this a *weak-paracomputer.*

A realization of the paracomputer (called the ultracomputer, see [S,80]) is one of the few currently feasible general purpose parallel machines. In foreseeable technologies, there is no shared memory, data items ar stored in local memories and a processor can receive or send only one data item per unit time. With this in mind, a simulation of an exclusive read - exclusive write machine (EREW) and that of a WRAM or a paracomputer are almost of the same cost (see [UW,84], [MV,83], [V,83b], [V,83a]). This justifies in part the use of a WRAM or stronger models for design of parallel algorithms.

The first of the problems we consider is that of *dynamic parallel addition*: n processors of a W-RAM are given a number x_i each, i = 1,...,n. It

is known that only $m \leq n$ of the x_i's are nonzero but processors do not know in advance which processors have the nonzero numbers. We show how to compute the sum of the x_i's in O(log m) expected parallel time and O(m) shared memory. The problem has applications in the construction of reliable arrays of sensors in robotics. To our knowledge, deterministic W-RAM algorithms for addition have to take O(log n) parallel time when n processors are used, and n numbers are to be added.

The second problem is that of *processor identification*: n processors of a weak-paracomputer are given, each keeping either a 0 or a 1. Each of the processors must find out which are the processors with the 1's. We assume that each shared memory location can fit a number of at most O(n) size. We show that there is an algorithm to solve the problem in O(n/logn) parallel time and just one shared memory location! This parallel time is optimal for the paracomputer. To show this, we use a nice technique due to Erdos and Renyi [ER,63] . The solution of the same problem requires $\Omega(n)$ time in a W-RAM, even if O(n) shared memory locations are available. We show how to make the algorithm constructive, by a probabilistic technique.

The third problem we consider is the problem of *approximate parallel counting*: n processors of a W-RAM are given a number x_i each, $i = 1,\ldots,n$ and x_i is either 0 or 1. Each processor wants to find whether the 1's are a majority. (Processors do not know which processors contain 1's.) We provide a probabilistic algorithm for answering a version of this problem, which uses O(1) expected parallel time and $O\left(\frac{n}{\log n}\right)$ shared memory.

The version of the problem we solve assumes that the number of x_i's equal to 1 is either $> \frac{n}{2}$ or $< \frac{n}{\log n}$.

All our probabilistic techniques have the following strong property: If T is the actual parallel time of our algorithm and E(T) is its expected value, then $\text{Prob}[T > k \cdot E(T)] \leq n^{-c}$, where k is any constant value, and $c > 1$ depends linearly on k and can be controlled by the algorithm implementer at the expense of an additional constant fraction of the shared memory used.

2. Dynamic Parallel Addition

2.1. The Algorithm

Let the array M represent the shared memory. Let $a >= 2$ be a positive integer constant. Let each processor be equipped with a local variable TIME, intended to keep the current parallel step. The algorithm has 3 stages.

Stage 1 (Initialization)

In one parallel step, processors initialize $a \cdot m + 2$ shared memory locations to zero, by executing: "Processor P_j writes a zero to M[j] , if $j \leq am + 2$." Then, they all execute $TIME_j \leftarrow 0$;

Stage 2 (Memory Marking)

Processor P_j
IF $x_j \neq 0$ then
BEGIN

(1) Select y equiprobably at random from {1,2,...,am}

(2) $TIME_j \leftarrow TIME_j + 1$

(3) Read M[y]; $TIME_j \leftarrow TIME_j + 1$

(4) If M[y] = 0 then write x_j into M[y] .
Also, $TIME_j \leftarrow TIME_j + 1$

(5) If the "write" failed then
BEGIN
(5a) write $TIME_j$ into M[am+1]
(5b) go to [1]
END
END
Comment: This part is executed by P_j with $x_j = 0$ and by "successful" P_j with $x_j \neq 0$.

(6) Read M[am+1] into a local variable R1

(7) Wait for 8 steps

(8) Read M[am+1] into a local variable R2.

(9) If R1 ≠ R2 then go to [6]

Comment: R1 = R2 means all processors with $x_j \neq 0$ succeeded into writing x_j in a shared memory location, different for each processor, among M[1] ,...,M[aM] . (If a processor was failing, the value of M[am+1] would change).

Stage 3 (Addition)
(Processor P_j is assigned to location M[j] , $1 \leq j \leq am$)

>From this point on, processors P_j (where $1 \leq j \leq am$) perform a standard parallel addition of the numbers M[1] ,...,M[aM] . In the ith parallel step of the addition, processor P_j adds M[j] and $M[j + 2^i]$ into M[j], for $j = k \cdot 2^i + 1$, $k = 0,1,2,...,am/2^i$. (See e.g. [K,82] or [FW,78] on how to do the parallel addition of am numbers by am processors in O(m) space and O(log m) parallel time.)

2.2. Analysis of the Algorithm

It is easy to see that the algorithm presented uses O(m) shared memory and performs the addition correctly, because, at the end of stage 2, the m nonzero x_j's are placed one in each of m shared memory locations, and these locations are among M[1] ,...,M[am] . The rest of these locations contain zeros.

The time complexity depends crucially on time of stage 2 (stage 1 takes O(1) parallel time and stage 3 takes O(log m) parallel time). It is easy to see that every time a processor P_j attempts to write its x_j, and if $g <= m$ shared memory locations are already "occupied", the competitors of P_j are m-g-1. Even if all of them manage to select different memory locations which were not occupied previously, the maximum number of locations that P_j must "avoid" is g + m-g-1 = m-1. So, P_j will succeed with probability at least $\lceil am-(m-1) \rceil /am \geq (a-1)/a$ in each trial (and this holds for every P_j). In the full paper we prove that:

Theorem 1. The average number of attempts required for all m processors to succeed is O(log m). The probability that the parallel time exceeds β log m is $\leq m^{-\beta \log a + 1}$ (and can be made arbitrarily small).

Proof sketch: The probability that there exists a processor which continues failing for $\geq \beta$ log m rounds is

$$\leq m(1/a)^{\beta \log m} \leq m^{-\beta \log a + 1}.$$

Corollary 1. Our algorithm performs dynamic parallel addition in O(log m) parallel expected time. It uses O(m) shared memory. The probability that the parallel time of the algorithm exceeds β log m is $\leq m^{-\beta \log a + 1}$.

3. The processor identification problem

3.1. An 0(n/log n) parallel time, O(1)-shared memory, Weak-Paracomputer Algorithm.

We provide here an algorithm which needs only 1 shared memory location, *which can hold only up to* O(n) *size-numbers.*[2]

Let us remark there that if a subset $S=P_{i_1}, \ldots, P_{i_s}$ of the n processors attempt simultaneously a FA(v,e) on the shared memory location v, and e is the value of the processor P_{i_k}, then, after the FA operation, the memory location shall be augmented by the sum s·e. Let us imagine that all the processors are equipped with the same list $L = \ell_1, \ell_2, \ldots, \ell_m$ of "testing" sequences, where each ℓ_i is an n-bit sequence of 0's and 1's. Let us assume

[2]In the *usual* paracomputer model, with *shared memory capable of holding* $O(2^n)$ *size numbers*, one can solve the problem in O(1) parallel time by having processors P_i execute $FA(v, 2^i \cdot e_i)$ where e_i is the value of Pi, and have each Pi read the result subsequently.

also that L is independent of the particular assignment of 0's or 1's to processors. In the following, let v be a fixed memory location, and let e_i be the value of processor Pi. Processors execute the following sequence of steps, m times.

Round i, $(1 \leq i \leq m)$

(1) P_1 erases v's contents by reading v and then doing FA(v,-value(v)).

(2) Each processor P_j $(1 \leq j \leq n)$ looks in the jth position of ℓ_i. If $\ell_i[j] = 1$ and $e_j = 1$, then P_j executes FA(v,e_j).

(3) Each P_j $(1 \leq j \leq n)$ reads v by executing FA(v,0).

At the end of the m rounds, each processor has, for each testing sequence, the *number* of places in which a 1 stands both in the testing sequence and in the sequence $e_1e_2 \cdots e_n$ to be guessed. If L allows each processor to find $e_1 \ldots e_n$ after the m rounds, we call L an m-*algorithm* for the processor identification problem. (We allow unrestricted local memory per processor.)

An obvious L (which would take O(n) parallel time) is that consisting of n ℓ_i's with $\ell_i[j] = 0$, $j \neq i$ and $\ell_i(i) = 1$ $(1 \leq i \leq n)$.

Erdos and Renyi [ER,63] considered a very closely related problem, the "coin-weight" problem. Using their techniques, we show that the m needed is $\theta(n/\log n)$ and that L is *constructive* (in fact, there is an easy probabilistic way to find L).

Let us view L as an $m \times n$ matrix of 0's and 1's.

Lemma 2 (see also [ER,63]). A matrix L, $m \times n$, of 0's and 1's is an m-algorithm for the processor identification problem iff: For each pair c,c' of subsets of the set C of columns of L, such that $c \neq c'$, if we form the row-sums of the submatrices L(c) and L(c') (consisting of the selected columns) and denote by v_c and $v_{c'}$ the column-vectors, consisting of these row-sums, then $v_c \neq v_{c'}$.

Proof sketch: After m-rounds, each processor has a row-sum vector, v, of L. This corresponds to *just one subset* c of the set of columns of L. This subset determines the processors which have 1's. This is so, because c is exactly the subset of processors with a value equal to 1.

Here, the techniques of [ER,63] can be employed, to prove:

Lemma 3. A matrix L, $m \times n$, $m = an/(\log n)$, $a \geq 6$, chosen so that the mn entries are independent random variables each taking on the values 0 and 1 with probability 1/2, is an m-algorithm, with probability tending to 1 as $n \to +\infty$.

Proof sketch (see full paper for details): Let p = prob{L is an m-algorithm}. Let q = 1-p. Let E (c_1,c_2), where c_1,c_2 are subsets of the set of columns C of L, denote the event that $v_{c_1} = v_{c_2}$ (v_{c_i} is the row-sum vector of $L(c_i)$). If c_1

and c_2 are not disjoint, then if $\tilde{c}_1$ is c_1 minus the set $c_1 \bigcap c_2$ and $\tilde{c}_2 = c_2 - c_1 \bigcap c_2$, then $\nabla_{\tilde{c}_1} = \nabla_{\tilde{c}_2}$. Hence, if L is not an m-algorithm, there exist *disjoint* subsets of the set of columns such that $\nabla_{c_1} = \nabla_{c_2}$. So

$$q \leq \sum \text{ProbE}(c_1, c_2)$$

where

$$c_1, c_2 \text{ disjoint subsets of } C.$$

By combinatorial calculations, one can get then

$$q \leq 2^{n(\log 3 - a/2) + o(n)}$$

where

$$m = \frac{an}{\log n.}$$

If we choose $a > \log_2 9 + 2$ then

$$q \leq 2^{-n} \text{ and } \lim_{n \to \infty} q = 0$$

Lemma 4. There exists an m-algorithm, for $m = \frac{an}{\log n}$, $a \geq 6$.

Proof sketch (see full paper for details): From Lemma 3, since $q < 1$ for such m *and any* n, then $p > 0$ so, in fact, there exists an m-algorithm.

3.2. Lower bounds and Conclusions

Lemma 5. No m-algorithm can have $m < \frac{n}{\log(n+1)}$.

Proof sketch: Each processor needs at least $\log(2^n) = n$ "pieces of information" to distinguish between the 2^n possible assignments of 0's and 1's to processors. On the other hand, if k processors attempt a FA, then the amount of information obtained cannot exceed $\log(k+1) \leq \log(n+1)$ because the number of 1's among them are 0,1, or, ..., or k. So, m simultaneous FAs can give each processor at most $m\log(n+1)$ pieces of information.

Note: Lemma 5 gives us lower bounds in the trade-off between the shared memory available and the number of parallel steps a weak-paracomputer needs to solve the problem. Also, note that instead of the weak-paracomputer we could just use a W-RAM (we call it the SW-RAM) with the property that simultaneous writes succeed only if they write the same value and, if that is the case, *their sum* is recorded. So, we get:

Theorem2. (A) The processor identification problem can be solved by a weak-paracomputer in O(n/log n) parallel-time, n processors and O(1) shared memory, and this is best possible.

(B) The same problem can be solved by n^2 processors in $O\left(\frac{n}{\text{slogn}}\right)$ time

and O(s) shared memory, and this is best possible.

Theorem 3. Any W-RAM algorithm requires at least $\Omega(n)$ parallel time to solve the processor identification problem, by n processors and O(1) shared memory.

(Proof in full paper).

and

Corollary 3. The weak paracomputer and the SW-RAM are more powerful models than the W-RAM.

Note: Once the processors have the list L (and L is an m-algorithm) they can construct a *table* of the possible row-sum vectors ∇_c and their corresponding subset c of L. Then, given any instance of the identification problem, they need O(m) parallel steps to find ∇_c and one (indexed) table access to find c (and solve the problem). Another piece of the preprocessing work is the construction of L itself. With O(1) shared memory and a weak paracomputer model, the n-processors will *need* $\theta(n^2/\log n)$ time to agree to a common (random) L. So, our algorithm becomes practical only in dynamic environments, where the values of the n processors change. Our algorithm can "identify" N sequences of 0's and 1's (where $N = \Omega(2^n)$) in preprocessing time $\Theta(\log^2 N/\log\log n)$, local space O(N), parallel time O (N log N/log log N and shared space O(1).

4. The problem of approximate counting

4.1. The problem and applications

We consider a CRCW WRAM of n processors $P_1,\ldots,P_n$. Processor Pi is given a number $x_i \in 0,1$. Processors know in advance that either the 1's are a majority ($> n/2$) or that their number is less than n/log n, but they don't know which are the Pi's with $x_i = 1$. We present here a technique of allowing the processors to distinguish one of the two situations, in O(1) expected parallel time, and O(n/log n) shared memory locations. We can restrict the shared memory so that each location can keep only 1 bit (i.e. O(n/log n) separately addressable *bits* of shared memory suffice). In the following, let M be the array of shared memory locations. One possible application of the result is parallel connectivity algorithms with a random graph as input. [ER,60] proved that random graphs with $\geq n$ edges and n vertices have a "giant" connected component (of size $\Theta(n)$) with probability tending to 1 as $n \to \infty$. Our techniques then allow the connectivity algorithm to identify the giant component without actually counting vertices (and hence beat the $\Omega(\log n)$ worst case bound in parallel time). In the following, let $a \geq 4$ be a constant and let $m = \lfloor n/(a \log n) \rfloor + 1$.

4.2. The algorithm

Processor Pi initializes M[i] to 0, $1 \leq i \leq m$.

(1) Each Pi, with $x_i = 1$, chooses equiprobably at random an integer $y \in \{1,2,\ldots,m-1\}$.

(2) Each Pi, with $x_i = 1$, writes its x_i to M[y].

(3) Each Pi, $1 \leq i \leq m-1$, reads M[i] . If M[i] = 0 then P_i writes 1 to M[m] .

(4) All P_i read M[m]. If it is 1, then they decide that the number of nonzero x_i's is $< n/\log n$ else they decide that it is $> \frac{n}{2.}$

4.3. Analysis of the algorithm

The algorithm of Section 4.2 clearly uses $\lfloor n/(a \log n) \rfloor + 1$ separately addressable units of shared memory (m *bits* would do) and takes O(1) parallel time. We now prove:

Theorem 4. Our algorithm decides correctly with probability $\geq 1 - n^{-c}$, $c > 1$ a constant.

Proof: Let $t = |\{P_i : x_i = 1\}|$. The algorithm errs in two cases:

Case 1. $t > n/2$ and M[m] = 1. This means that at least n/2 processors selected an integer among 1 and n/(a log n) at random and one number was not selected. The probability that a particular location was *not* selected is $(1 - (a \log n)/n)^{n/2} \approx n^{-a/2})$. The probability that there exists a location which is not selected is, then, upper bounded by

$$n \cdot n^{-a/2} = n^{-a/2+1} = n^{-c_1},$$

where

$$c_1 = a/2 - 1 \geq 1$$

since $a \geq 4$.

Case 2. $t < n/\log n$ and M[m] = 0. This means that less than n/log n processors selected an integer among 1 and n/(a log n) at random and all integers were selected. The situation is similar to that throwing N = n/log n balls at random in N/a boxes and having at the end at least 1 ball per box. From [F,58] the probability of the above event is asymptotically $e^{-n/a}$ (tending to 0 as $n \to \infty$). The theorem follows.

4.4. Discussion

By repeating the algorithm of sec. 4.2 we can get an algorithm which never errs and whose *expected* parallel time is O(1). A special case of our algorithm could give a probabilistic, fast, n-input NAND implementation. We can refine the algorithm so that the "interval of uncertainty" becomes much less than [n/log n, n/2] , with small penalty in the expected parallel time (see full paper).

REFERENCES

[CLC,83]
Chin, F., J. Lam and I. Chen, "Optimal Parallel Algorithms for the Connected Components Problem," CACM83.

[C,80]
Cook, S., "Towards a Complexity Theory of Synchronous Parallel Computations", Specker Symp. on Logic and Algorithms, Zurich, Feb.5-11, 1980.

[DNS,81]
Dekel, E., D. Nassimi and S. Sahni, "Parallel Matrix and Graph Algorithms," SIAM J.Comp. 10(4) 1981.

[ER,60]
Erdos, P. and A. Renyi, "On the Evolution of Random Graphs," The Art of Counting, J.Spencer Editor, MIT Press, 1973.

[ER,63]
Erdos, P. and A. Renyi, "On two problems of information theory", Mayar Tud. Akad. Mat. Kut. Int. Kozl. 8(1963); also in The Art of Counting, J.Spencer, Editor, MIT Press, 1973.

[G,78]
Goldschlager, L., "A Unified Approach to Models of Synchronous Parallel Machines", Proc. 11 th sub STOC, May 1978.

[G,77]
Goldschlager, L., "Synchronous Parallel Computation", Ph.D. thesis, Univ.of Toronto, C.S. Dept., 1977.

[GLR,80]
Gottlieb, A., B. Lubachevsky and L. Rudolph, "Basic Techniques for the efficient coordination of very large numbers of cooperating sequential processors," Courant Inst. TR No.028, Dec.1980.

[HCS,79]
Hirschberg, D., A. Chandra, D. Sarwate, "Computing Connected Components on Parallel Computers," CACM 22(8) Aug.1979.

[K,82]
Kucera, L., "Parallel Computation and Conflicts in Memory Access", Info. Processing Letters Vol.14, April 1982.

[MV,83]
Melhorn, K., and U. Vishkin, "Randomized and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories," 9th Workshop on Graph Theoretic Concepts in Computer Science, Univ. Usnabruck, June 1983.

[R,82]
Reif, J., "Symmetric Complementation," 14th STOC, San Francisco, CA, May 1982.

[R,82b]
Reif, J., "On the Power of Probabilistic Choice in synchronous Parallel Computations", 9th ICALP, Aarchus, Denmark, July 1982.

[Ru,79]
Ruzzo, W., "On Uniform Circuit Complexity", Proc.20th FOCS, Oct.1979.

[SJ,81]
Savage, C. and J. Ja'ja', "Fast, Efficient Parallel Algorithms for Some Graph Problems", SIAM J.Comp. 10(4), Nov.1981.

[SV,80]
Shiloach, Y. and U. Vishkin, "Finding the Maximum Merging and Sorting in a Parallel Computation Model", Tech. Rep. Technion Israel, Comp. Sci., March 1980.

[SV,82]
Shiloah, Y. and U. Vishkin, "An O(logn) Parallel Connectivity Algorithm", J.of Algorithms, 1982.

[S,80]
Schwartz, J.T., "Ultracomputers", ACM TOPLAS 1980, pp.484-521.

[UW,84]
Upfal, E. and A. Wigderson, "How to share memory in a distributed system", 25th FOCS, Proceedings, October 1984.

[V,83a]
Vishkin, U., "A parallel-design, distributed-implementation general purpose parallel computer", to appear, J.TCS.

[V,83b]
Vishkin, U., "Randomized speeds-ups in parallel computation", 16th STOC, April 1984, Proceedings.

[U,84]
Upfal, E., "A probabilistic relation between desirable and feasible models of parallel computation", 16th ACM STOC 1984, Proceedings.

[W,79]
Wyllie, J., "The Complexity of Parallel Computation", Ph.D. Thesis, Cornell University, 1979.

NYU COMPSCI TR-144c.1
Spirakis, Paul G

Fast probabilistic
techniques for ...

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012